Analysis and Comparison of Brute-Force and Genetic Algorithm methods on Traveling Salesman Problem

Onur Temizkan
Department of Computer Engineering, Izmir Institute of Technology, Urla / Izmir / Turkey

Abstract: ***Traveling Salesman Problem (TSP) is a common NP-Complete problem in Computer Science and Optimization. Thus, there is no known way to solve it in polynomial time at the time of this writing. In this report, Genetic algorithm approach will be analyzed on TSP and compared with Brute-Force Search method.***

***Index Terms*—traveling salesman problem, optimization, genetic algorithm, TSP, combinatorics.**

I. Introduction

In this report, The performance of a genetic algorithm implementation will be analyzed, onto Traveling Salesman Problem to find the optimum path to traverse a multi-node undirected and weighted graph. On the other side, to measure the pros / cons of the Genetic Algorithms, brute-force method will also be applied on the same problem. The problem definitions and solutions are implemented in Python.

II. Problem Definition

For this research, the problem is defined as a *undirected graph* with *N* nodes. There are edges with several weights between those nodes. In TSP analogy, the nodes represent cities, and edges represent paths between those cities. Edge weights, represent the distances between two cities. A route is valid if the salesman start traveling from a given node, visit all cities once, and return back to the starting city.

Since the graph is not necessarily a *complete graph*, the algorithm should check next available nodes from the node at a single time to create the route. All nodes should be visited once. So the implementation should be aware of the possible circular routes and should avoid them.

Moreover, since the graph edges are weighted by definition, the algorithms we implement should select from possible routes in awareness of costs.

The TSP that will be solved is shown in Figure 1

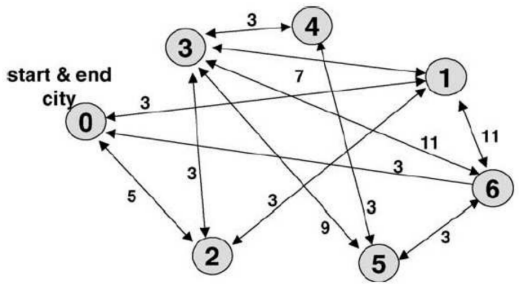

Fig. 1. TSP to be solved

III. Methods

In this experiment, two different methods will be applied to the same problem.

A. Brute-Force Search

Brute-Force Search solves this problem in an inefficient but a definite way. To find the best route, Brute-Force Search checks all possible path permutations without any specific condition or restriction. While checking routes, it stores and updates the shortest route from the start of the execution. When the checks are finished, the shortest route is the last route stored. Since it checks all possible routes, it makes sure that the returned result is the best possible solution. This implementation is very memory-efficient because only thing that is stored, is the shortest path and its size. But it's also very time inefficient because all the path combinations are tried.

B. Genetic Algorithm

Solving the problem with a Genetic Algorithm is more efficient but indefinite. This is because in a Genetic Algorithms, it's not necessary to check all possible routes. Instead, evolving a set of routes is tried to create a better route. GA approach is an iterative method which is based on the natural selection mechanism of evolution. Since natural selection *-thus evolution* doesn't have a stopping point, Genetic Algorithms do not have a strict finishing point. So it's presumed that there can always be a better solution at the any state of the algorithm execution, and a stopping condition should be chosen.

Algorithm 1 Genetic Algorithm on TSP

1. Create a population with *N* random routes.
2. Select the best *k* routes
3. combine / match the selected routes to create an ancestor route.
4. Mutate that ancestor node *N-1* times to create a new generation with N new routes.
Repeat steps *2* to *4*,
Until the best route hasn't been changing for x generations
5. Return the best route and its length.

IV. Implementation

The code [1] of both Brute-Force and Genetic Algorithm approaches are implemented in Python. We tried to keep the code simple by removing any nonessential parts such as status logs. This also helped to measure and compare the performances since there are almost no avertible computational cost in any of those two implementations.

Python definition and representation of the TSP is shown below in Figure 2. Each member of `PATH_DEF`, represent a city with its connections to other cities. The connections are represented by a list of tuples. The first element of each tuple represents the connected city, and the second one represents the distance to that city.

```
PATH_DEF = {
    0: [(1, 3), (2, 5), (6, 3)],
    1: [(0, 3), (2, 3), (3, 7), (6, 11)],
    2: [(0, 5), (1, 3), (3, 5)],
    3: [(1, 7), (2, 3), (4, 3), (5, 9), (6, 11)],
    4: [(3, 3), (5, 3)],
    5: [(3, 9), (4, 3), (6, 3)],
    6: [(0, 3), (1, 11), (3, 11), (5, 3)]
}
```

Fig. 2. TSP Data Definition

An excerpt from Brute-Force implementation is given below. This implementation first find all the possible routes in the problem definition, then calculates the path sizes of each route. While doing that, it stores the shortest route at the time of execution. After traversing all the routes, it returns the best route found and its distance.

```
def trav_salesman():
    for point in PATH_DEF:
        find_all_paths([point])

    min_length = inf
    min_path = None

    for path in all_paths:
        path_length = calc_path_length(path)
        if path_length < min_length:
            min_path = path
            min_length = path_length
```

Fig. 3. Brute Force Implementation

Genetic algorithm implementation is a little bit more complex. It first creates a random population. Then until a limit of generations, it generates a new populations using fittest paths. An excerpt from that implementation can be found in Figure 4.

Since our test example is an incomplete undirected graph, our genetic implementation is based on specifically for that kind of graphs, another implementation for complete graphs can also be found in given repository. [1]

```
def trav_salesman():
    population = pre_populate()

    for i in range(MAX_GENERATIONS):
        fittest = get_fittest(population)
        (fittest_path, fittest_length) = fittest

        all_generation_fittest_lengths.append(
            fittest_length
        )
        all_generation_average_lengths.append(
            get_average(population)
        )
        all_generation_worst_lengths.append(
            get_worst(population)[1]
        )

        population = populate(fittest_path)
```

Fig. 4. Genetic Algorithm Implementation

V. Results

1) First Trial

In the first implementations of these two algorithms, the results were interesting. Genetic Algorithm took more time than Brute-Force implementation which was unexpected. The results are shown in Table I.

	Brute Force	Genetic Algorithm
Iters / Gens	102	7
Path	[0, 6, 5, 4, 3, 2, 1]	[3, 2, 1, 0, 6, 5, 4]
Distance	18	18
Time (sec.)	0.00134	0.0024

TABLE I
Compared Execution Results of First Implementation

The reason of this would be the mutation logic and sample size selection. Also since the used problem size was very small, that implementation performed worse than brute-force implementation.

The Genetic Algorithm implementation had a mutation logic which created a significant computational overhead. This overhead may become negligible in larger size problems but in this case, traversing all possible routes in Brute-Force implementation became less costly than these logic in Genetic Algorithm logic.

Also as will be stated in V-2, this implementation didn't have any 'Crossover' logic, which might be the source of the performance problem.

So, the implementation should had been changed. What should had been tried is, keeping the mutation logic -*of course* but making it more efficient for the algorithm to finish with less generations, thus earlier.

2) Second Trial

A new implementation with updated sample size and changed mutation logic, gave better results.

The things that had changed were instead of mutating the best route in each generation while creating the new generation, the best N generations are combined to create the new one, this implementation is actually the correct implementation of Genetic Algorithms. In the first implementation 'Crossover', which is an important step of Genetic Algorithms had been skipped. This implementation had Crossover step and kept the mutation logic.

The problem about the Crossover implementation on this problem was, since the graph is not fully connected, the result of crossover / and its mutations could be invalid routes. After applying crossover and also mutations, a validation had needed. If a route hadn't pass those validations, crossovers or mutations had to be repeated until finding a valid route.

Although this implementation seems to have more computational overhead, it finds the shortest path in less generations. So this implementation worked much better than its predecessor.

The new results are shown in Figure 5 and Table II.

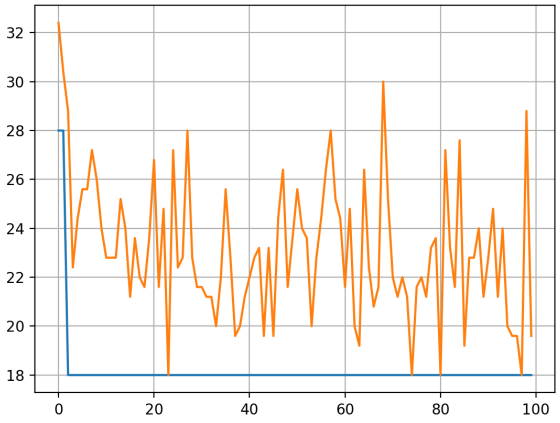

Fig. 5. New results showing best route sizes in a generation in blue, and mean size of a generation in orange.

	Brute Force	Genetic Algorithm
Iters / Gens	102	2
Path	[0, 6, 5, 4, 3, 2, 1]	[3, 2, 1, 0, 6, 5, 4]
Distance	18	18
Time (sec.)	0.00134	0.00037

TABLE II
COMPARED EXECUTION RESULTS OF SECOND IMPLEMENTATION

As can be seen, the new implementation of Genetic Algorithm Solution, the execution time got smaller and the result is found in the second generation. Also interestingly, the found solution of the Genetic Algorithm implementation is exactly the same as the first implementation results.

VI. CONCLUSION

Genetic Algorithm approach works very well on Traveling Salesman Problem. That approach significantly reduced the number of visited routes to find the optimal route, comparing with the Brute-Force Search. Although, our sample problem is not so big, it reduced the execution time in a significant amount. When the problem gets bigger, the execution time benefits can be much more significant.

Also, as stated in V, the implementation is very important for the performance of the Genetic Algorithm. The performance measures can differ a lot with simple optimizations on implementation decisions.

Giving directions to a searching quest of an optimal path should be almost every time better than just searching blindly. But in the sense of Genetic Algorithms the given directions seems to be even more accurate, providing that the correct solution is found in just two generations in our second implementation in V-2.

To summarize, given the results of this experiment, Genetic Algorithms seem to be good solutions on NP-Complete problems. But since this experiment doesn't compare Genetic Algorithms with other possible solvers such as 'Linear Programming', 'Nearest Neighbour', 'Simulated Annealing' and others, what is the best approach for these kind of problems cannot be stated.

Note that this implementation is only a solution for incomplete graph based TSPs. For other TSP domains like coordinate systems, other implementations and logic should be applied. A sample implementation of these kind of problems can also be found in [1].

REFERENCES

[1] http://github.com/onurtemizkan/tsm-genetic-algorithm